

Termination of Monotone Programs

Omar Al-Bataineh¹, Xie Xiaofei¹, Alwen Tiu¹, and Mark Reynolds²

¹Nanyang Technological University, Singapore

²The University of Western Australia

Abstract. We present an efficient approach to prove termination of monotone programs with integer variables, an expressive class of loops that is often encountered in computer programs. Our approach is based on a lightweight static analysis method and takes advantage of simple properties of monotone functions. Our preliminary implementation shows that our tool has an advantage over existing tools and can prove termination for a high percentage of loops for a class of benchmarks.

1 Introduction

Proving termination of programs is a challenging and important problem whose solutions can significantly improve software reliability and programmer productivity. Termination analysis also plays a role in the analysis of reactive systems, non-terminating systems that engage in ongoing interaction with their environments. In this case, a termination argument is used to prove liveness properties such as the absence of deadlock or livelock, by establishing that some desired behaviour is not postponed forever. While the problem of proving termination has now been extensively studied and has been shown undecidable [28,25,7,4,18,8,16,17,21,20,29,3,23], search for efficient methods for proving non-termination that can cover a broad range of programs remains open.

A huge body of work has been done on proving termination of programs that is based on a number of techniques such as abstract interpretation [4,8,29], bounds analysis [16,17], ranking functions [6,14], recurrence sets [18,20] and transition invariants [21,25]. The most popular technique to prove termination is through the synthesis of a ranking function, a mapping from the state space to a well-founded domain, whose value monotonically decreases as the computation progresses. However, it is known that the linear ranking approach cannot completely resolve the problem [24,7], since there are terminating programs that have no such ranking function [3]. Moreover, linear ranking functions do not suffice for all loops, and, in particular for multipath loops [2]. Thus, the problem of decidability of termination for linear loops stays open, in its different variants.

Monotonicity is a useful property that is often encountered in computer programs [19,27]. Indeed, a very high percentage of variables of programs are monotonic in nature and therefore the analysis of such programs can be significantly improved by exploiting monotonicity. The complexity of the termination problem together with the observation that most loops in practice have (relatively)

simple termination argument and are monotone in nature suggest the use of light-weight static analysis for this purpose.

The termination problem is a hard problem, and one can expect it to be decidable only in the simplest cases [7]. In this work we study the termination problem of a simple class of loops with multiple paths and linear monotone assignments. The loop is specified by some initial condition, a loop guard, and a “loop body” of some restricted form. We allow the loop guard to be either a conjunction of diagonal-free constraints (i.e. where the only allowed comparisons are between a loop variable and a constant) or diagonal constraints (where difference between two variables can be also compared with constants). The analysis takes advantage of properties of monotone expressions [19,27], where we use recurrence relations to model the loop iteration update of each loop variable over a single variable $n \in \mathbb{N}$ denoting the loop counter. We consider a wide variety of loop forms with different syntactic structure (i.e. different forms of loop guard and update expressions). For each of these forms we identify exactly the concrete conditions under which the loop can be said to be non-terminating. These conditions turn out to have a surprising level of complexity. In particular, we find that these conditions can differ, depending on the assumptions that we make about the syntactic structure of the loop. An advantage of the proposed approach is that it works on programs as they are, without imposing any extra annotation effort or invoking a safety checker as other methods typically do.

To demonstrate the efficiency of the proposed approach we compare our implementation against the (strongest) tools that participate in SV-COMP¹ using two benchmarks: the SNU real-time benchmark (www.cprover.org/goto-cc/examples/snu.html) and the Power-Stone benchmark [22]. Namely, we compare the implementation against the tools AProVE [1] and 2ls [9]. Our implementation outperforms both AProVE and 2ls in two aspects. First, it can handle a class of monotone programs that AProVE and 2ls fail to handle. Second, for the programs that all tools can handle our tool can decide termination much faster.

Related Work. Termination is a fundamental decision problem in program verification. In particular, termination of programs with linear assignments and guards has been extensively studied over the last decade. This has led to the development of powerful semi-algorithms to prove termination via synthesis of ranking functions, several of which have been implemented in software-verification tools such as T2 [13] and ARMC [26]. The work in [10,11,6] use algorithms with sub-procedure for ranking individual paths; which is focused on the iterative construction of a termination argument for a full program. See the surveys by Ben-Amram and Genaim [2], and by Gasarch [15] describing semi-algorithmic approaches to termination based on ranking functions.

In [28] Tiwari proves that the termination of a class of single-path loops with linear guards and assignment is decidable when the domain of the variables is \mathbb{R} . Braverman proved that this holds for \mathbb{Q} as well [7]. However, both

¹ <http://sv-comp.sosy-lab.org/2016/>

have considered universal termination of single-path loops, the termination of single/multi-path loops for a given input left open.

Biere, Artho, and Schuppan propose an encoding of liveness properties into an assertion [5]. This approach allows proving termination of programs without a ranking sub-procedure. It has been reported to prove termination of programs that require non-linear ranking functions. Prior experimental results on some benchmarks indicate this encoding results in difficult safety checks [12].

Berdine et al. present an algorithm for proving termination that is based on abstract interpretation [4]. Using an invariance analysis they construct a variance analysis, and they use the fact that the transitive closure of a well-founded relation is also well-founded to show that the fixed-point obtained by their analysis is correct.

2 Preliminaries

In this section we introduce some definitions that we use throughout the paper.

Definition 1. *A linear loop \mathcal{L} with integer variables x_1, \dots, x_n is a while loop of the form*

$$\textbf{while } (\phi_1 \wedge \dots \wedge \phi_m) \textbf{ do } \{s_1; \dots; s_n\}$$

where each condition ϕ_i is one of the following form $(x_i \sim c)$ or $((x_i - x_j) \sim c)$ such that $c \in \mathbb{Z}$ and $\sim \in \{<, \leq, >, \geq\}$. We call the constraints of the form $(x_i \sim c)$ as diagonal-free constraints and the constraints of the form $((x_i - x_j) \sim c)$ as diagonal constraints. Each instruction s_i is one of the following form

$$s_i := \text{Assignment} \mid \textbf{if } \phi \textbf{ then Assignment else Assignment}$$

where ϕ is a condition and Assignment can be of the following form

$$x_i := u \mid x_i := u * x_i \mid x_i := x_i + v \mid x_i := u * x_i + v$$

such that $u, v \in \mathbb{Z}$.

Intuitively, the semantics of a while loop can be interpreted as follows: starting from initial values for the variables x_1, \dots, x_n (the input), the instructions s_1, \dots, s_n are executed either sequentially or conditionally as far as the condition $(\phi_1 \wedge \dots \wedge \phi_m)$ holds. We say that the loop terminates for a given input if the condition $(\phi_1 \wedge \dots \wedge \phi_m)$ eventually evaluates to *false*.

In this paper we are interested in studying termination of linear monotone loops. The property of monotonicity of a statement is defined with respect to a specific loop surrounding the statement. Consider a while loop \mathcal{L} and a statement $\mathbf{s} : x := e$; inside the loop. Further consider a single execution of the loop which involves n iterations through the loop. Let $\ell_1, \ell_2, \dots, \ell_n$ denote the n consecutive iterations of the loop and x_1, x_2, \dots, x_m , denote the values assigned to x during these iterations, where $m \leq n$ because statement \mathbf{s} may not be executed during every iteration if there are conditional branches inside the loop.

Definition 2 (Monotonic statements [27]). *A statement \mathbf{s} is considered to be loop monotonic w.r.t. loop \mathcal{L} if the sequence of values assigned to variable x during successive executions of \mathbf{s} forms an increasing or decreasing sequence of values (i.e. $x_i < x_{i+1}$ or $x_i > x_{i+1}$). The monotonic statement \mathbf{s} is considered to be regular monotonic if the sequence x_1, x_2, \dots, x_m is an arithmetic progression or geometric progression. Otherwise, the monotonic statement is considered to be irregular monotonic.*

The class of monotonicity of a statement $\mathbf{s} : x := e$; surrounded by a loop \mathcal{L} can be determined using a set of static techniques. In [27] a sophisticated static analysis is employed in order to determine loop monotonic variables. For space reasons we omit discussion of these techniques and we refer the reader to [27]. Throughout this paper we use the following notations: \uparrow to denote a monotonically increasing update expression, \downarrow to denote a monotonically decreasing update expression, and \rightarrow to denote a constant expression.

3 Single-path Linear Monotone Loops

In this section we consider termination of single-path linear monotone loops. We consider first termination of single-path loops with diagonal-free constraints and then single-path loops with diagonal constraints. The structure of single-path loops with diagonal-free constraints that we consider has the following form

$$\mathbf{while}(x \sim c) \ \mathbf{do} \ \{x := \mathbf{f}(x); \} \quad (1)$$

where $\sim \in \{<, \leq, >, \geq\}$, $c \in \mathbb{Z}$, and the statement $\mathbf{s} : x := \mathbf{f}(x)$; is a monotonic statement. Termination of a program of the form (1) is very straightforward and can be decided using Lemma 1.

Lemma 1. *Let \mathcal{L} be a program of the form (1). Then \mathcal{L} is terminating if $\sim \in \{<, \leq\}$ and $\mathbf{s} \uparrow$ or $\sim \in \{>, \geq\}$ and $\mathbf{s} \downarrow$. On the other hand, \mathcal{L} is non-terminating if $\sim \in \{<, \leq\}$ and $\mathbf{s} \downarrow$ or $\sim \in \{>, \geq\}$ and $\mathbf{s} \uparrow$, given that the initial value of x (let us call it x_0) satisfies the loop guard.*

3.1 Single-path Loops with Diagonal Constraints

In this section we consider single-path loops with diagonal constraints of the form

$$\mathbf{while} \ ((x - y) \sim c) \ \mathbf{do} \ \{x := \mathbf{f}_1(x); \ y := \mathbf{f}_2(y); \} \quad (2)$$

where $\sim \in \{<, \leq, >, \geq\}$, $c \in \mathbb{Z}$, and the statements $\mathbf{s}_1 : x := \mathbf{f}_1(x)$; and $\mathbf{s}_2 : y := \mathbf{f}_2(y)$; are monotonic statements.

In Table 1 we classify linear monotone expressions into regular and irregular expressions, where regular expressions are classified further into regular

Update expression	Regularity class	Recurrence equation	Notation
$x := x + v, v \neq 0$	Regular with arithmetic progression	$x_n = x_0 + v * n$	R_a
$x := u * x, u > 1$	Regular with geometric progression	$x_n = x_0 * u^n$	R_g
$x := u * x + v, u > 1, v \neq 0$	Irregular	$x_n = u^n * x_0 + \sum_{i=0}^{n-1} (u^i * v)$	I

Table 1: A summary of classification of linear monotone expressions

expressions with arithmetic progression and regular expressions with geometric progression. We model the loop iteration update of each loop variable as a recurrence equation over a new variable $n \in \mathbb{N}$ denoting the loop counter.

Recurrence equations are fundamental models used for the definition of relations between consecutive elements of a sequence. In this work, we show that for linear monotone programs, it is more natural to represent the behaviour of loop variables and to reason about termination using recurrence equations instead of using conventional approaches such as linear ranking approach. Using such formalization, the analysis can take advantage of fundamental mathematical properties of various types of recurrence relations which arise during the analysis. As shown in Table 1 there are two types of recurrence equations that arise during the analysis of monotone programs: linear and exponential equations.

We now discuss different procedures to analyse termination of a program of the form (2) depending on the various types of recurrence relations which arise during the analysis. We consider the case where $\sim \in \{>, \geq\}$ since the analysis of the case where $\sim \in \{<, \leq\}$ is very similar. Note that if the variables move in opposite directions (i.e. one increases and the other decreases) then the analysis is trivial (i.e. if x moves up and y moves down then the program is non-terminating, while if x moves down and y moves up then the program is terminating, given that the initial values (x_0, y_0) satisfy the loop guard). However, if the variables move in the same direction then the analysis can be non-trivial.

When both variables grow/decay linearly of the form R_a . Let s_1 and s_2 be monotonic statements of the form R_a . Termination of a program of the form (2) can be then decided using the following rules.

1. If $(s_1 \uparrow \wedge s_2 \uparrow \wedge (v_1 \geq v_2))$ then the program is non-terminating. If $(s_1 \uparrow \wedge s_2 \uparrow \wedge (v_1 < v_2))$ then the program is terminating.
2. If $(s_1 \downarrow \wedge s_2 \downarrow \wedge (|v_1| \leq |v_2|))$ then the program is non-terminating. If $(s_1 \downarrow \wedge s_2 \downarrow \wedge (|v_1| > |v_2|))$ then the program is terminating.

When both variables grow/decay exponentially of the form R_g . Let s_1 and s_2 be monotonic statements of the form R_g . Termination of a program of the form (2) can be then decided using the following rules.

1. If $(s_1 \uparrow \wedge s_2 \uparrow \wedge (u_1 \geq u_2))$ then the program is non-terminating. If $(s_1 \uparrow \wedge s_2 \uparrow \wedge (u_1 < u_2))$ then the program is terminating.
2. If $(s_1 \downarrow \wedge s_2 \downarrow \wedge (|u_1| \leq |u_2|))$ then the program is non-terminating. If $(s_1 \downarrow \wedge s_2 \downarrow \wedge (|u_1| > |u_2|))$ then the program is terminating.

When one variable grows/decays linearly and the other exponentially.

Let \mathbf{s}_1 be a monotonic statement of the form R_a and \mathbf{s}_2 be a monotonic statement of the form R_g or I . Then termination can be decided using the following rules.

1. If $(\mathbf{s}_1 \uparrow \wedge \mathbf{s}_2 \uparrow)$ then the program is terminating since for any exponential function and any linear function with positive growth, the exponential function will eventually outstrip the linear function.
2. If $(\mathbf{s}_1 \downarrow \wedge \mathbf{s}_2 \downarrow)$. This case requires special decision procedure (see Algorithm 1) since it is not obvious whether there will be an iteration at which the guard $(x_n - y_n \sim c)$ can be violated.

Input: $(x_0, y_0, v_1, v_2, u, c, \sim)$
output: $\{\text{terminating}, \text{non-terminating}\}$
int $n := 1$;
while (*true*)
 {
 $x_n = (x_0 + v_1 * n); \quad y_n = u^n * y_0 + \sum_{n=0}^{n-1} (u^n * v_2);$
 if $((\sim = '>' \wedge y_n \geq (x_n - c)) \vee (\sim = ' \geq ' \wedge y_n > (x_n - c)))$ **return** *terminating*;
 if $(y_n < (x_n - c) \wedge x_n < 0 \wedge y_n < 0)$ **return** *non-terminating*;
 $n := n + 1$;
 }

Algorithm 1: Special decision procedure for case 2

Note that when algorithm 1 terminates with ‘terminating’ then the search has reached an iteration where the loop guard is violated. On the other hand, when the algorithm terminates with ‘non-terminating’ then the search has reached an iteration where $(y_n < (x_n - c) \wedge x_n < 0 \wedge y_n < 0)$. This condition is sufficient to guarantee non-termination of this form of loops since x is decreasing linearly while y is decreasing exponentially.

We now turn to discuss the case where x grows/decays exponentially and y grows/decays linearly. In this case termination can be decided as follows.

1. If $(\mathbf{s}_1 \downarrow \wedge \mathbf{s}_2 \downarrow)$ then the program is terminating since there will be an iteration where x can be smaller than y . This is due to the assumption that x is decreasing exponentially while y is decreasing linearly.
2. If $(\mathbf{s}_1 \uparrow \wedge \mathbf{s}_2 \uparrow)$. This case requires a special decision procedure (similar to Algorithm 1 above) but it differs at the condition at which it returns ‘non-terminating’ which will be of the form $(x_n > (y_n + c))$. This is due to the observation that once the exponential function outstrips the linear function at some iteration it continues to outstrip it at all future iterations.

When both variables grow/decay exponentially. In this case a decision procedure similar to Algorithm 1 can be used to decide termination. The procedure differs from Algorithm 1 at the condition at which it terminates with

‘non-terminating’. That is, if both variables are moving up then the procedure terminates with ‘non-terminating’ when the search reaches an iteration at which $((x_n > y_n + c) \wedge u_1 \geq u_2)$ since both x and y grow exponentially as the loop proceeds. On the other hand, if both variables are moving down then the condition at which the algorithm terminates with ‘non-terminating’ will be of the form $(y_n < (x_n - c) \wedge x_n < 0 \wedge y_n < 0 \wedge u_1 \leq u_2)$ since both variables decay exponentially as the loop proceeds.

In Table 2 we summarise the stopping condition at which a decision procedure of the form 1 returns ‘non-terminating’ when the variables x and y move in the same direction but with different mathematical behaviour. Note that the procedures return ‘terminating’ when the loop guard $(x - y \sim c)$, where $\sim \in \{>, \geq\}$, is violated but they differ at the stopping condition at which they return ‘non-terminating’ since this depends on the characteristics of the recurrence update equations of x and y (i.e. whether they grow/decay linearly or exponentially). The $+$, $-$ signs used as a superscript of notations R_a , R_g , and I to indicate whether the variable is moving up (+) or moving down (-).

Update statements	Stopping condition (non-termination)
(1) $\mathbf{s}_1 = R_a^- \wedge \mathbf{s}_2 = R_g^-$ (2) $\mathbf{s}_1 = R_a^- \wedge \mathbf{s}_2 = I^-$	$(y_n < (x_n - c) \wedge x_n < 0 \wedge y_n < 0)$
(3) $\mathbf{s}_1 = R_g^+ \wedge \mathbf{s}_2 = R_a^+$ (4) $\mathbf{s}_1 = I^+ \wedge \mathbf{s}_2 = R_a^+$	$(x_n > (y_n + c))$
(5) $\mathbf{s}_1 = R_g^+ \wedge \mathbf{s}_2 = I^+$ (6) $\mathbf{s}_1 = I^+ \wedge \mathbf{s}_2 = R_g^+$	$((x_n > y_n + c) \wedge u_1 \geq u_2)$
(7) $\mathbf{s}_1 = R_g^- \wedge \mathbf{s}_2 = I^-$ (8) $\mathbf{s}_1 = I^- \wedge \mathbf{s}_2 = R_g^-$	$(y_n < (x_n - c) \wedge x_n < 0 \wedge y_n < 0 \wedge u_1 \leq u_2)$

Table 2: Stopping condition at which the procedure returns non-terminating

4 Multi-path Loops with Diagonal-free Constraints

Paths of multi-path loops can interleave in a non-trivial manner. Reasoning about termination of multi-path loops is a therefore challenging task. The class of multi-path loops that we consider has the following form

$$\mathbf{while} (x \sim c) \ \mathbf{do} \ \{\mathbf{if} (x \sim' c_1) \ \mathbf{then} \ x := \mathbf{f}_1(x); \ \mathbf{else} \ x := \mathbf{f}_2(x); \} \quad (3)$$

where the statements $\mathbf{s}_1 : x := \mathbf{f}_1(x)$; and $\mathbf{s}_2 : x := \mathbf{f}_2(x)$; are monotonic statements, $\sim, \sim' \in \{<, \leq, >, \geq\}$, and $c, c_1 \in \mathbb{Z}$. We refer to the assignment $x := \mathbf{f}_1(x)$ as a conditional assignment, as its execution depends on the truth value of the condition $(x \sim' c_1)$. As we shall see in this section the complexity of termination analysis of a program of the form (3) varies depending on the class of monotonicity of the statements \mathbf{s}_1 and \mathbf{s}_2 and whether the conditional branches

of the program interleave between iterations. During the analysis of the different cases of a program of the form (3) we use ϕ to refer to the loop guard ($x \sim c$) and B to refer to the condition ($x \sim' c_1$). We write \sim_ϕ to refer to the relation operator used in the loop guard, \sim_B to refer to the relation operator used in the if-branch, and $\sim_{\neg B}$ to refer to the relation operator used in the else-branch.

Observation 1 *If the statements s_1 and s_2 in program (3) have the same monotonicity behaviour (i.e. both are monotonically increasing or monotonically decreasing) then we can safely ignore B and use Lemma 1 to verify termination.*

4.1 When One Statement is Monotone and the other is Constant

In this section we analyse termination of a program of the form (3) under the assumption that one of the update statements is strictly monotone and the other is constant. In this case there are several sub-cases to consider depending on the operators \sim , \sim' and the class of monotonicity of the statements s_1 and s_2 .

1. When $\sim_\phi \in \{>, \geq\}$ (i.e. x is bounded from below in loop guard ϕ) and $\sim_B \in \{<, \leq\}$ (i.e. x is bounded from above in B) and $s_1 \uparrow$ and $s_2 \rightarrow_b$ (i.e. $f_2(x) := b$, where b is a constant). From the syntactic structure of the given loop form and by observing the way the loop variable x is changed w.r.t to the loop condition it is easy to see that for this form of loops there is only one condition of non-termination. This happens when $(b \sim_\phi c)$, where \sim_ϕ is the relational operator used in the guard ϕ . Note that the if-branch can not contribute towards the termination of the loop since s_1 is monotonically increasing and x is bounded from below in ϕ . However, from the given loop form we note that when the if-branch is triggered then the else-branch will be triggered at some later iteration. Hence, the formula of non-termination for this form of loops can be described as follows

$$NT = (x_0 \models \phi \wedge (b \sim_\phi c)) \quad (4)$$

where x_0 denotes the initial value of x and $x_0 \models \phi$ means that the guard ϕ is evaluated to *true* when $x = x_0$. If formula (4) is satisfiable then we say the loop is non-terminating. Note that formula (4) can be used also to verify termination of loops in which $\sim_\phi \in \{<, \leq\}$, $\sim_B \in \{>, \geq\}$, $s_1 \downarrow$, and $s_2 \rightarrow_b$ for the same reasoning given above.

2. When $\sim_\phi \in \{>, \geq\}$ and $\sim_B \in \{<, \leq\}$ and $s_1 \downarrow$ and $s_2 \rightarrow_b$. For this form of loops there is only one case of non-termination. This case happens when the else-branch is triggered during the execution of the loop and that $(b \sim_\phi c)$ and $(b \sim_{\neg B} c_1)$. Note that if the if-branch is triggered during the execution of the loop then the loop will eventually terminate since x is bounded from below in ϕ and from above in B and s_1 is monotonically decreasing.

$$NT = (x_0 \models \phi \wedge x_0 \not\models B \wedge (b \sim_\phi c) \wedge (b \sim_{\neg B} c_1)) \quad (5)$$

Formula (5) can be used also to verify termination of loops in which $\sim_\phi \in \{<, \leq\}$, $\sim_B \in \{>, \geq\}$, $s_1 \uparrow$, and $s_2 \rightarrow_b$ for the same reasoning given above.

3. When $\sim_\phi \in \{<, \leq\}$ and $\sim_B \in \{<, \leq\}$ and $s_1 \uparrow$ and $s_2 \rightarrow_b$. Analysing termination of this form of loops is non-trivial since the conditional branches of the loop can interleave between iterations. However, before discussing the conditions of non-termination of this form of loops we need to observe the following. From the syntactic structure of the given loop form we note that when the loop is non-terminating and the if-branch is triggered then the else-branch will be triggered at some later iteration since x is bounded from above in B and s_1 is monotonically increasing. On the other hand, when the else-branch is triggered then the if-branch may or may not be triggered at future iterations depending on the value of the constant b . Hence, finding the conditions of non-termination of this form of loops requires some numerical analysis. Since x is bounded from above in B and s_1 is monotonically increasing then we need to compute the smallest value of x that falsifies the condition B when the if-branch is triggered with $x = d$, where $d \in \{x_0, b\}$. That value can be computed using some special numerical procedures depending on the class of monotonicity of the statement s_1 . That is, if s_1 is R_a (i.e. has the form $x := x + v$) then one can use formula (6). On the other hand, if s_1 is R_g or I then one needs to use Algorithm 2, where the bound c_1 that appears in the algorithm is the bound that x is compared to in the condition B .

$$\psi_a(d) = \begin{cases} ((c_1 + v) - ((c_1 - d) \% v)) & \text{if } \sim_B = '<=' \\ (((c_1 - 1) + v) - ((c_1 - 1) - d) \% v)) & \text{if } \sim_B = '<' \end{cases} \quad (6)$$

Let us explain the intuition behind formula (6). Note that at each iteration the if-branch is triggered the variable x is incremented by v . Hence, the maximum value that x can reach from the consecutive execution of the if-branch is either $(c_1 + v)$ or $((c_1 - 1) + v)$ depending on \sim_B . Therefore, if $(c_1 - d)$ is a multiple of v then the smallest value of x that falsifies B will be either $(c_1 + v)$ or $((c_1 - 1) + v)$. On the other hand, if $(c_1 - d)$ is not a multiple of v then the remainder needs to be subtracted from the maximum value $(c_1 + v)$ or $((c_1 - 1) + v)$ depending on \sim_B .

Input: (d, c_1, u, v, \sim_B)
int $n := 1$; $x_{n-1} := d$;
while ($true$)
 {
 $x_n = (u * x_{n-1} + v)$; $n++$;
 if ($(\sim_B = '<' \wedge x_n \geq c_1) \vee (\sim_B = '<=' \wedge x_n > c_1)$) **return** x_n ;
 }

Algorithm 2: Special procedure ψ_g, ψ_I

By analysing the conditional branches of the loop and their possible interleaving one can see that there are four conditions of non-terminating for this form of loops, which can formalised as follows

$$\begin{aligned}
NT = & ((x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee \\
& (x_0 \models \phi \wedge x_0 \models B \wedge \psi(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee \\
& (x_0 \models \phi \wedge x_0 \models B \wedge \psi(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_B c_1 \wedge \psi(b) \sim_\phi c) \vee \\
& (x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_B c_1 \wedge \psi(b) \sim_\phi c))
\end{aligned} \tag{7}$$

where $\psi(x_0)$ returns the smallest value of x at which the if-branch becomes disabled after being triggered using the value x_0 , and $\psi(b)$ returns the smallest value of x at which the if-branch becomes disabled after being triggered using the value b . Note that we write $\psi(x)$ in the formula which can be either ψ_a or ψ_g or ψ_I depending on the class of monotonicity of the statement s_1 .

4. When $\sim_\phi \in \{<, \leq\}$ and $\sim_B \in \{<, \leq\}$ and $s_1 \downarrow$ and $s_2 \rightarrow_b$. For this form of loops there are two cases of non-termination. The first case is when the if-branch is triggered during the execution of the loop. In this case the loop will not terminate since x is bounded from above in both ϕ and B and s_1 is monotonically decreasing. The second case is when the else-branch is triggered and that $(b \sim_\phi c)$. In this case the loop will not terminate regardless of which branch is triggered between iterations.

$$NT = ((x_0 \models \phi \wedge x_0 \models B) \vee (x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c)) \tag{8}$$

Formula (8) can be used also to verify termination of loops in which $\sim_\phi \in \{>, \geq\}$ and $\sim_B \in \{>, \geq\}$ and $s_1 \uparrow$ and $s_2 \rightarrow_b$.

5. When $\sim_\phi \in \{>, \geq\}$ and $\sim_B \in \{>, \geq\}$ and $s_1 \downarrow$ and $s_2 \rightarrow_b$. Termination analysis of this form of loops is similar to case 3 above in the sense that deriving the conditions of non-termination requires performing some numerical analysis. The only difference here is the formula that is used to compute the smallest value of x that falsifies the condition B when the if-branch is triggered at some prior iteration with $x = d$, where $d \in \{x_0, b\}$, since s_1 in this case is monotonically decreasing. This value can be computed using formula (9) in case s_1 is R_a (i.e. has the form $x := x + v$) or using a special procedure similar to Algorithm 2 in case s_1 is R_g or I .

$$\psi'_a(d) = \begin{cases} ((c_1 - v) + ((d - c_1) \% v)) & \text{if } \sim_B = '>\\ \\ (((c_1 + 1) - v) + ((d - (c_1 + 1)) \% v)) & \text{if } \sim_B = '\geq' \end{cases} \tag{9}$$

Similar to case 3 above there are four conditions of non-termination of this form of loops which can be formalized as given in formula (10).

$$\begin{aligned}
NT = & ((x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee \\
& (x_0 \models \phi \wedge x_0 \models B \wedge \psi'(x_0) \sim c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee \\
& (x_0 \models \phi \wedge x_0 \models B \wedge \psi'(x_0) \sim_\phi c \wedge b \sim c \wedge b \sim_\phi c_1 \wedge \psi'(b) \sim_\phi c) \vee \\
& (x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_B c_1 \wedge \psi'(b) \sim_\phi c))
\end{aligned} \tag{10}$$

4.2 When Both Update Statements are Strictly Monotone

This is the most challenging form of loops to consider since at some iteration of the loop the variable x may be updated using the arbitrary monotone statement \mathbf{s}_1 while in some later iteration it may be updated using the arbitrary monotone statement \mathbf{s}_2 . Hence, to analyse termination of this form of loops we need to consider several sub-cases depending on the class of monotonicity of the statements \mathbf{s}_1 and \mathbf{s}_2 and the operators \sim and \sim' .

1. When $\sim_\phi \in \{>, \geq\}$ and $\sim_B \in \{>, \geq\}$ and $\mathbf{s}_1 \uparrow$ and $\mathbf{s}_2 \downarrow$. Surprisingly, this case is trivial since there is only one condition under which the loop can be non-terminating as described in formula (11).

$$NT = (x_0 \models \phi \wedge x_0 \models B) \quad (11)$$

That is, if the if-branch is triggered in the first iteration of the loop then the loop is non-terminating since \mathbf{s}_1 is monotonically increasing and x is bounded from below in both ϕ and B . Note that the else-branch cannot contribute towards the non-termination of the loop since \mathbf{s}_2 is monotonically decreasing and x is bounded from below in ϕ and from above in $\neg B$. Formula (11) can be used also to verify termination of loops in which $\sim_\phi \in \{<, \leq\}$, $\sim_B \in \{<, \leq\}$, $\mathbf{s}_1 \downarrow$, and $\mathbf{s}_2 \uparrow$ for the same reasoning given above.

2. When $\sim_\phi \in \{<, \leq\}$ and $\sim_B \in \{<, \leq\}$ and $\mathbf{s}_1 \uparrow$ and $\mathbf{s}_2 \downarrow$. For this form of loops it is not possible to derive a concrete formula for verifying termination as we have done for the previous cases since in case of non-termination the two branches will activate each other in alternation and hence the loop variable will be updated using any of the two arbitrary statements. We therefore use a special procedure to analyse termination of this form of loops. The procedure (see Algorithm 3) is guaranteed to return with a correct answer. The procedure uses the auxiliary formulas $\psi(x)$ and $\psi'(x)$ to compute the precise value of x between iterations. The procedure maintains a list called PASSED to store the values of x that are already examined. Note that if at some iteration of the loop the guard ϕ is violated then the loop is terminating. On the other hand, if the search reaches a fixed point (i.e. the same value of x is encountered again during the search) then the loop is non-terminating. Reaching a fixed point in case of non-termination is guaranteed since the domain of x at both branches is finite. Note also that for this form of loops the if-branch is the only branch that can contribute towards the termination of the loop as x is bounded from above in ϕ and \mathbf{s}_1 is monotonically increasing. This is the reason why we check for termination only at the if-branch.
3. When $\sim_\phi \in \{<, \leq\}$ and $\sim_B \in \{>, \geq\}$ and $\mathbf{s}_1 \uparrow$ and $\mathbf{s}_2 \downarrow$. For this form of loops there is only one case of non-termination. This case happens when the else-branch is triggered at the first iteration of the loop. Since x is bounded from above in ϕ and from above in $\neg B$ and \mathbf{s}_2 is monotonically decreasing then the loop will not terminate in this case. Note that the if-branch cannot contribute towards the non-termination of the loop since x is bounded from below in B and from above in ϕ and \mathbf{s}_1 is monotonically increasing.

```

Input:  $(x_0, \phi, B)$ 
output:  $\{terminating, non-terminating\}$ 
PASSED :=  $\emptyset$ ; int VAL =  $x_0$ ;
while (true)
  if (VAL  $\not\models B$ ) then if  $(\psi'(VAL) = x_i)$  for any  $x_i \in$  PASSED then
    return 'non-terminating' else {add VAL to PASSED; VAL :=  $\psi'(VAL)$ }
  if (VAL  $\models B$ ) then if  $(\psi(VAL) = x_i)$  for any  $x_i \in$  PASSED then
    return 'non-terminating' else if  $(\psi(VAL) \not\models \phi)$  then
      return 'terminating' else {add VAL to PASSED; VAL :=  $\psi(VAL)$ }

```

Algorithm 3: Special decision procedure for case 2 at Section 4.2

$$NT = (x_0 \models \phi \wedge x_0 \not\models B) \quad (12)$$

Note that formula (12) can be used also to verify termination of loops in which $\sim_\phi \in \{>, \geq\}$ and $\sim_B \in \{<, \leq\}$ and $s_1 \downarrow$ and $s_2 \uparrow$.

4. When $\sim_\phi \in \{>, \geq\}$ and $\sim_B \in \{<, \leq\}$ and $s_1 \uparrow$ and $s_2 \downarrow$. This case is similar to case 2 above since in case of non-termination the conditional branches will interleave between iterations in a non-trivial manner. The only difference here is that the if-branch in this case cannot contribute towards the termination of the loop while the else-branch can do so.

```

Input:  $(x_0, \phi, B)$ 
output:  $\{terminating, non-terminating\}$ 
PASSED :=  $\emptyset$ ; int VAL =  $x_0$ ;
while (true)
  if (VAL  $\models B$ ) then if  $(\psi(VAL) = x_i)$  for any  $x_i \in$  PASSED then
    return 'non-terminating' else {add VAL to PASSED; VAL :=  $\psi(VAL)$ }
  if (VAL  $\not\models B$ ) then if  $(\psi'(VAL) = x_i)$  for any  $x_i \in$  PASSED then
    return 'non-terminating' else if  $(\psi'(VAL) \not\models \phi)$  then
      return 'terminating' else {add VAL to PASSED; VAL :=  $\psi'(VAL)$ ;}

```

Algorithm 4: Special decision procedure for case 4 at Section 4.2

In Table 3 we summarize the termination rules of all possible cases of multi-path loops with diagonal-free constraints of the form (3), where c represents the bound that the loop variable is compared to in ϕ , and c_1 is the bound that the variable is compared to in the condition B . Since the relational operator used in the guard ϕ and in the condition B can be either bounded from below $\{>, \geq\}$ or bounded from above $\{<, \leq\}$ and that the update statements s_1 and s_2 can be either \uparrow or \downarrow or \rightarrow_b , then we have a total of 36 cases to consider as shown in the table. We now discuss some simple examples of loop programs to demonstrate how the termination rules described in Table 3 can be used to decide termination.

Example 1. Consider the program:

```

while( $x \geq 5$ ) {if ( $x \geq 10$ ) then  $x := x + 1$ ; else  $x := x - 1$ ; }

```

Loop form	Termination rule
(1) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \uparrow \wedge s_2 \rightarrow_b)$ (2) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \downarrow \wedge s_2 \rightarrow_b)$ (3) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \rightarrow_b \wedge s_2 \uparrow)$ (4) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \rightarrow_b \wedge s_2 \downarrow)$	$(x_0 \models \phi \wedge (b \sim_\phi c))$
(5) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \downarrow \wedge s_2 \rightarrow_b)$ (6) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \uparrow \wedge s_2 \rightarrow_b)$	$(x_0 \models \phi \wedge x_0 \not\models B \wedge (b \sim_\phi c) \wedge (b \sim_{\neg B} c_1))$
(7) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \rightarrow_b \wedge s_2 \downarrow)$ (8) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \rightarrow_b \wedge s_2 \uparrow)$	$(x_0 \models \phi \wedge x_0 \models B \wedge (b \sim_\phi c) \wedge (b \sim_{\neg B} c_1))$
(9) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \rightarrow_b \wedge s_2 \uparrow)$ (10) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \rightarrow_b \wedge s_2 \downarrow)$	$((x_0 \models \phi \wedge x_0 \not\models B) \vee (x_0 \models \phi \wedge x_0 \models B \wedge b \sim_\phi c))$
(11) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \downarrow \wedge s_2 \rightarrow_b)$ (12) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \uparrow \wedge s_2 \rightarrow_b)$	$((x_0 \models \phi \wedge x_0 \models B) \vee (x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c))$
(13) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \uparrow \wedge s_2 \rightarrow_b)$	$((x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \models B \wedge \psi(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \models B \wedge \psi(x_0) \sim c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi(b) \sim_\phi c) \vee$ $(x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi(b) \sim_\phi c))$
(14) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \rightarrow_b \wedge s_2 \downarrow)$	$((x_0 \models \phi \wedge x_0 \models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \not\models B \wedge \psi(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \not\models B \wedge \psi(x_0) \sim c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi(b) \sim_\phi c) \vee$ $(x_0 \models \phi \wedge x_0 \models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi(b) \sim_\phi c))$
(15) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \downarrow \wedge s_2 \rightarrow_b)$	$((x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \models B \wedge \psi'(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \models B \wedge \psi'(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi'(b) \sim_\phi c) \vee$ $(x_0 \models \phi \wedge x_0 \not\models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi'(b) \sim_\phi c))$
(16) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \rightarrow_b \wedge s_2 \downarrow)$	$((x_0 \models \phi \wedge x_0 \models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \not\models B \wedge \psi'(x_0) \sim_\phi c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1) \vee$ $(x_0 \models \phi \wedge x_0 \not\models B \wedge \psi'(x_0) \sim c \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi'(b) \sim_\phi c) \vee$ $(x_0 \models \phi \wedge x_0 \models B \wedge b \sim_\phi c \wedge b \sim_{\neg B} c_1 \wedge \psi'(b) \sim_\phi c))$
(17) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \uparrow \wedge s_2 \downarrow)$ (18) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \downarrow \wedge s_2 \uparrow)$	$(x_0 \models \phi \wedge x_0 \models B)$
(19) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \uparrow \wedge s_2 \downarrow)$ (20) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \downarrow \wedge s_2 \uparrow)$	$(x_0 \models \phi \wedge x_0 \not\models B)$
(21) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \uparrow \wedge s_2 \downarrow)$ (22) $(\sim_\phi \in \{<, \leq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \downarrow \wedge s_2 \uparrow)$	Special decision procedure (see Algorithm 3)
(23) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{<, \leq\} \wedge s_1 \uparrow \wedge s_2 \downarrow)$ (24) $(\sim_\phi \in \{>, \geq\} \wedge \sim_B \in \{>, \geq\} \wedge s_1 \downarrow \wedge s_2 \uparrow)$	Special decision procedure (see Algorithm 4)
(25-28) $(\sim_\phi, \sim_B \in \{<, \leq, >, \geq\} \wedge s_1 \rightarrow_{b_1} \wedge s_2 \rightarrow_{b_2})$	$((x_0 \models B \wedge b_1 \sim_\phi c \wedge b_1 \sim_{\neg B} c_1) \vee (x_0 \not\models B \wedge b_2 \sim_\phi c \wedge b_2 \sim_{\neg B} c_1) \vee$ $(b_1 \sim_\phi c \wedge b_2 \sim_\phi c))$
(29-32) $(\sim_\phi, \sim_B \in \{<, \leq, >, \geq\} \wedge s_1 \uparrow \wedge s_2 \uparrow)$ (33-36) $(\sim_\phi, \sim_B \in \{<, \leq, >, \geq\} \wedge s_1 \downarrow \wedge s_2 \downarrow)$	See Observation 1

Table 3: The termination rules of all possible cases for loop programs of the form (3)

with $x_0 = 15$. Note that the syntactic structure of the given program matches with case 17 in Table 3 and hence we need to use the rule $(x_0 \models \phi \wedge x_0 \models B)$ to verify termination. As one can see the rule holds for the given program and hence the loop is non-terminating.

Example 2. Consider the program:

while $(x \leq 10)$ **{if** $(x \leq 5)$ **then** $x := x + 2$; **else** $x := x - 3$; **}**

with $x_0 = 3$. For this program we will use Algorithm 3 to verify termination. Note that after five iterations (i.e. $x = 3, 5, 7, 4, 6, 3$) the search will reach a fixed point with $x = 3$ and hence the program is non-terminating.

5 Experimental Results

We compare our tool with two state-of-the-art tools: (1) AProVE [1] which is a system for automated termination and complexity proofs of term rewrite systems

(TRSs), and (2) 2LS [9] a CPROVER-based framework, which reduces program analysis problems expressed in second order logic such as invariant or ranking function inference to synthesis problems over templates. In the 5th Competition on Software Verification (SV-COMP’16), AProVE was the strongest tool for the termination category, while 2LS has been shown to be a powerful tool for proving termination for larger programs with thousands of lines of code [9]. We evaluate the tools using a large number of ANSI-C programs including:

- The SNU real-time benchmark suite that contains small C programs used for worst-case execution time analysis which are available at www.cprover.org/goto-cc/examples/snu.html;
- The Power-Stone benchmark suite as an example set of C programs for embedded systems [22].

During the evaluation we make the following manual simplifications: (1) we move the loops in the programs to the main function, (2) we simplify the complex statements and functions that do not affect the termination of the loop (e.g., make the function only contains one return statement), and (3) we add some variables declaration and initialization to ensure the correctness of the program. Our simplifications do not affect the outcome of termination of programs.

All experiments were run on Ubuntu 14.10, 3.2 GHZ Intel Core 4 duo CPU with 8GB RAM. The results for SNU and Power-Stone are presented in Tables 4 and 5. Each table reports the number of loops that were proven as terminating (T), non-terminating (NT), time-out (TO), or Maybe (M). Note that we set the timeout value to 300s. In most cases, the tools decide within a few seconds. As we observe in almost every program the proposed approach requires cheaper computational effort to verify termination. It is interesting to mention that for the analysed programs of the SNU benchmark suite, AProVE could handle only 23 programs while 2LS could handle 45 programs. On the other hand, for the Power-Stone benchmark suite, AProVE could handle only 30 programs while 2LS could handle 53 programs. More detailed information on the results and performance of the tools is available at <https://sites.google.com/site/termresult>.

Our implementation outperforms both AProVE and 2ls in two aspects. First, it can handle a class of monotone programs that AProVE and 2ls fail to handle. Second, for the programs that all tools can handle our tool can decide termination much faster. The key advantage of the proposed approach is that it uses a light-weight static analysis technique based on recurrence equations to reason about termination, while the other tools use synthesis approaches based on ranking techniques. Finding a ranking function of a loop program and proving its correctness is often expensive. The tools need to infer ranking function and invariants on demand by iterating the loops. The analysis shows that tools based on ranking techniques may fail to find a ranking function even for simple monotone loop programs as shown in Tables 4 and 5. During the analysis, we note also that the tool AProVE does not handle very well programs with complex statements (i.e. those which contain pointers, arrays, and other data structures) even that the update statements that affect termination are simple monotone statements. For such cases the tool generates “Maybe” as an output.

Programs\Tools	Our tool		AProVE		2ls	
	result	time(ms)	result	time(s)	result	time(s)
adpcm_loop12.c	T	1.173	T	6.18	T	0.76
fft1_loop6.c	T	1.186	M	1.12	T	0.40
minver_loop2.c	T	1.601	M	1.10	T	0.17
crc_loop3.c	T	1.765	T	54.27	M	0.42
minver_loop6.c	T	2.209	M	1.10	T	0.23
fft1_loop1.c	T	0.904	M	1.18	T	0.18
fft1_loop4.c	T	0.915	T	2.41	T	0.18
minver_loop7.c	T	1.795	M	1.16	T	0.20
fft1_loop5.c	T	1.493	M	1.14	TO	300.06
adpcm_loop11.c	T	0.955	T	2.69	T	0.48
adpcm_loop9.c	T	1.306	T	2.19	T	0.17
fft1_loop2.c	T	0.848	M	1.34	T	0.19
fft1_loop3.c	T	1.121	M	1.07	M	0.77
matmul_loop1.c	T	1.375	M	2.89	T	0.22
crc_loop1.c	T	1.318	T	10.28	T	0.16
adpcm_loop5.c	T	1.091	T	9.17	T	0.56
minver_loop1.c	T	1.666	M	1.18	T	0.15
fft1k_loop4.c	T	1.807	M	1.04	M	9.01
adpcm_loop17.c	T	1.342	M	1.17	T	1.27
minver_loop9.c	T	2.118	M	1.19	T	0.53
lms_loop6.c	T	1.137	M	1.12	M	0.36
adpcm_loop16.c	T	1.286	M	1.23	T	1.15
ludcmp_loop4.c	T	8.527	M	1.32	T	0.41
lms_loop7.c	T	1.093	M	1.24	T	0.18
ludcmp_loop5.c	T	1.178	M	1.10	TO	300.06
fft1k_loop2.c	T	0.848	T	1.98	T	0.18
adpcm_loop8.c	T	0.978	M	2.01	T	0.12
adpcm_loop1.c	T	1.003	T	2.07	T	0.11
ludcmp_loop1.c	T	1.568	M	1.29	TO	300.02
fir_loop1.c	T	0.928	T	1.97	T	0.13
minver_loop3.c	T	2.326	M	1.10	T	0.43
adpcm_loop10.c	T	1.630	T	2.80	T	0.14
adpcm_loop15.c	T	1.364	M	1.23	M	3.44
adpcm_loop2.c	T	1.209	T	1.98	T	0.14
adpcm_loop14.c	T	1.229	T	22.24	T	0.24
adpcm_loop3.c	T	1.241	T	8.87	T	0.63
qsort_exam_loop1.c	T	1.882	M	1.13	M	0.28
minver_loop4.c	T	2.138	M	1.14	T	0.23
adpcm_loop13.c	T	1.032	T	2.51	T	0.16
minver_loop5.c	T	1.169	M	2.27	T	0.39
minver_loop8.c	T	1.781	M	1.11	T	0.27
adpcm_loop7.c	T	1.133	T	5.29	T	0.13
fir_loop5.c	T	1.195	M	1.06	M	0.82
matmul_loop2.c	T	1.412	M	8.78	T	0.30
fir_loop2.c	T	0.899	T	2.13	T	0.18
lms_loop4.c	T	1.330	M	1.14	TO	300.22
lms_loop5.c	T	1.146	M	1.15	TO	300.22
adpcm_loop8.c	T	1.374	T	6.31	T	0.13
fft1k_loop1.c	T	0.916	T	1.95	T	0.21
fdctint_loop1.c	T	0.981	T	2.47	T	0.39
adpcm_loop6.c	T	1.024	M	2.41	T	0.17
fft1k_loop3.c	T	2.066	M	1.21	TO	300.02
fir_loop4.c	T	1.355	M	1.15	M	1.08
ludcmp_loop2.c	T	1.194	M	1.28	T	3.73
fir_loop3.c	T	1.046	M	1.17	TO	300.22
ludcmp_loop3.c	T	1.177	M	1.15	TO	300.04
fdctint_loop2.c	T	1.919	TO	300.06	T	0.83
fdctint_loop3.c	T	9.560	TO	300.05	T	0.74
lms_loop2.c	T	0.898	T	2.14	T	0.17
lms_loop1.c	T	1.074	T	1.98	T	0.13
fibcall_loop1.c	T	0.995	T	44.49	M	6.24
lms_loop3.c	T	1.261	M	1.10	M	0.85
crc_loop2.c	T	1.612	TO	300.05	T	0.33
Total	63	99.102	23	1154.1	45	2442.33

Table 4: SNU real-time benchmark suite

Programs\Tools	Our tool		AProVE		2ls	
	result	time(ms)	result	time(s)	result	time(s)
adpcm_loop12.c	T	2.494	M	13.64	T	0.29
blit_loop3.c	T	1.244	M	2.01	T	0.14
compress_loop6.c	T	1.312	TO	319.34	T	0.22
huff_loop1.c	T	1.221	T	4.85	T	0.19
compress_loop7.c	T	1.569	M	8.84	T	26.41
compress_loop8.c	T	1.117	M	1.68	T	18.54
pocsag_loop2.c	T	2.110	M	1.13	T	0.66
adpcm_loop11.c	T	1.787	TO	306.37	T	0.93
huff_loop2.c	T	1.500	T	16.92	T	1.46
blit_loop1.c	T	1.188	M	1.25	T	0.49
adpcm_loop9.c	T	1.695	T	2.57	T	0.48
huff_loop3.c	T	1.375	M	4.36	T	6.30
g3fax_loop1.c	T	1.108	T	6.53	T	1.50
compress_loop4.c	T	1.120	T	2.89	T	22.48
compress_loop1.c	T	1.149	T	2.38	T	1.04
jpeg_loop10.c	T	1.056	T	2.47	T	0.18
fft_loop1.c	T	1.352	T	9.84	T	0.17
compress_loop5.c	T	1.101	T	2.02	T	12.30
huff_loop4.c	T	1.718	T	4.51	T	2.77
compress_loop2.c	T	1.034	M	2.93	T	0.58
compress_loop3.c	T	1.045	M	6.07	T	0.88
ucbqsort_loop4.c	T	0.960	M	1.43	M	0.13
adpcm_loop4.c	T	1.938	M	2.18	T	0.13
adpcm_loop1.c	T	1.622	T	10.14	T	0.25
adpcm_loop10.c	T	1.873	M	1.14	T	2.15
adpcm_loop5.c	T	1.593	T	5.50	T	0.15
ucbqsort_loop3.c	T	0.981	M	1.24	M	0.15
adpcm_loop2.c	T	1.480	M	1.82	T	0.18
ucbqsort_loop2.c	T	1.056	M	1.39	M	0.13
adpcm_loop3.c	T	1.468	T	22.39	T	0.29
v42_loop5.c	T	1.515	M	1.06	TO	300.71
ucbqsort_loop1.c	T	1.420	T	6.51	T	0.56
v42_loop3.c	T	0.941	T	1.98	T	0.19
jpeg_loop3.c	T	1.343	M	2.20	T	0.12
pocsag_loop1.c	T	1.075	T	1.83	T	0.15
pocsag_loop4.c	T	1.479	T	2.43	T	0.57
jpeg_loop2.c	T	1.265	T	9.07	T	0.21
jpeg_loop7.c	T	1.277	M	2.23	T	0.13
pocsag_loop5.c	T	1.788	T	1.49	T	0.23
jpeg_loop6.c	T	1.110	T	2.08	T	0.19
v42_loop2.c	T	0.952	T	1.75	T	0.17
fir_loop2.c	T	1.758	M	1.24	M	1.68
adpcm_loop7.c	T	1.404	T	2.21	T	0.17
jpeg_loop1.c	T	1.163	T	2.60	T	0.24
adpcm_loop8.c	T	1.439	T	2.85	T	0.12
v42_loop4.c	T	1.518	M	1.20	TO	300.35
adpcm_loop6.c	T	1.518	T	4.96	T	0.15
fir_loop1.c	T	1.148	M	1.24	TO	300.31
pocsag_loop3.c	T	0.974	T	1.81	T	0.11
jpeg_loop11.c	T	1.033	T	7.35	T	10.35
fir_loop3.c	T	1.191	M	1.15	M	1.38
blit_loop2.c	T	1.450	M	1.26	T	0.37
jpeg_loop8.c	T	2.366	T	11.33	T	21.05
huff_loop5.c	T	1.836	M	9.61	T	1.86
pocsag_loop6.c	T	1.535	M	5.07	T	0.41
v42_loop6.c	T	1.440	M	1.94	M	1.39
pocsag_loop8.c	T	2.197	M	1.17	T	0.45
v42_loop1.c	T	1.328	T	18.66	T	0.15
jpeg_loop5.c	T	1.148	T	6.28	T	0.14
jpeg_loop4.c	T	1.363	M	2.66	T	0.20
jpeg_loop9.c	T	1.182	T	2.57	T	3.45
pocsag_loop7.c	T	1.246	M	7.05	T	0.27
crc_loop2.c	T	1.576	T	61.93	M	0.58
Total	63	88.244	30	958.64	53	1051.98

Table 5: PowerStone benchmark suite

6 Conclusion and Future Work

We presented an efficient approach to prove termination of monotone programs, a class of loops that is often encountered in computer programs. The approach uses a light-weight static analysis technique based on recurrence equation and takes advantage of properties of monotone functions. The proposed approach leads to significant performance improvement against previous tools where they are applicable and it can handle a class of monotone programs that AProVE and 2ls fail to handle. In future work, we aim to extend the tool to support a richer class of monotone programs, in particular we aim to study termination of multi-path programs that are more general than the ones considered in this paper, but for which termination is still decidable.

References

1. Proving termination of programs automatically with aprobe. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR*, pages 184–191, 2014.
2. Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *J. ACM*, 61:26:1–26:55, 2014.
3. Amir M. Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. *ACM Trans. Program. Lang. Syst.*, 34:16–24, 2012.
4. Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O’Hearn. Variance analyses from invariance analyses. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’07, pages 211–224, New York, NY, USA, 2007. ACM.
5. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *FMICS02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*. Elsevier, 2002.
6. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV’05, pages 491–504, Berlin, Heidelberg, 2005. Springer-Verlag.
7. Mark Braverman. Termination of integer linear programs. In *Computer-Aided Verification*, CAV, pages 372–385, 2006.
8. Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, pages 148–162. Springer-Verlag, 2008.
9. Hong-Yi Chen, Daniel Kroening, Peter Schrammel, and Bjoern Wachter. Synthesising interprocedural bit-precise termination proofs. In *ASE*, pages 53–64, 2015.
10. Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 67–81, London, UK, UK, 2001. Springer-Verlag.
11. Michael Colón and Henny Sipma. Practical methods for proving program termination. In *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 442–454, 2002.
12. Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 6015 in LNCS, page 236250. Springer, 2010.
13. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’06, pages 415–426. ACM, 2006.
14. Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, pages 1–24, 2005.
15. William Gasarch. Proving Programs Terminate Using Well-Founded Orderings, Ramsey’s Theorem, and Matrices. *Advances in Computers*, 97:147–200, 2015.
16. Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.

17. Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL09*, 2009.
18. Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 147–158. ACM, 2008.
19. Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 272–282. ACM, 1990.
20. William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 304–319, 2010.
21. Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, 2010.
22. Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 389–392. ACM, 2007.
23. Jan Leike and Matthias Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11, 2015.
24. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251. Springer, 2004.
25. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 32–41, 2004.
26. Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *In PADL*. Springer, 2007.
27. Madalene Spezialetti and Rajiv Gupta. Loop monotonic statements. *IEEE Trans. Softw. Eng.*, pages 497–505, 1995.
28. A. Tiwari. Termination of linear programs. In *Computer-Aided Verification, CAV*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004.
29. Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *TACAS*, volume 6605. Springer, 2011.